Applying Loop Tiling and Unrolling to a Sparse Kernel Code

E. Herruzo[1], G. Bandera[2], and O. Plata[2]

[1] Dept. Electronics, University of Córdoba, Spain
[2] Dept. of Computer Architecture, University of Málaga, Spain

Abstract. Code transformations to optimize the performance work well where a very precise data dependence analysis can be done at compile time. However, current compilers usually do not optimize irregular codes, because they contain input dependent and/or dynamic memory access patterns. This paper presents how we can adapt two representative loop transformations, tiling and unrolling, to codes with irregular computations, obtaining a significant performance improvement over the original non-transformed code. Experiments of our proposals are conducted on three different hardware platforms. A very known sparse kernel code is used as an example code to show performance improvements.

1 Introduction

Over the years, the ratio between the main memory latency and processor cycle time has been increasing. Computer architects have proposed several hardware mechanisms that reduce the impact of the memory latency problem: lockup-free caches, prefetching, out-of-order execution, etc... The efficiency of architectural improvements depends on the compiler ability to change the structure of programs for taking full advantage of them.

When optimizing a program an important performance improvement will come from optimizing the most time-consuming code regions, that is, repetitive sentence blocks. In this way, a number of compiler strategies have been developed to enhance the performance: (1) strategies that change the original data layout of the array variables [4,3]; (2) strategies based on loop restructuring transformations that reduce the number of executed instructions and/or change the order in which statements are executed [1]. Some works also attempt to integrate both strategies as a single algorithm [2].

Most of the compiler optimizations were designed for regular computations [5]. If access patterns are input or code conditional dependent, compiler optimizations are much more difficult to decide and apply. There are many special memory access patterns, that appear frequently in irregular applications, where no data dependence analysis is needed in order to apply some optimization transformations. The majority of compilers, however, do not take into account these special situations, loosing the opportunity of obtaining a better object code.

This paper analyzes one of the most important special irregular access patterns, resulting from the multiplication of a sparse matrix by a dense array

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3036, pp. 409–412, 2004.

(spMxV). We will show that various commercial compilers, from Compaq, Silicon Graphics and Cray, do not optimize codes with this kind of computational structure. However, manually applying powerful optimization techniques, a significant performance improvement is obtained.

2 Optimizing Sparse Codes

Conventional data dependence techniques are not usually applicable to irregular codes, due to the variant nature of the reference patterns. This is one of the main reasons why compilers usually cannot decide to apply loop optimizations and then the obtained object code is frequently sub-optimal. In this section we will analyze two widely used loop transformation techniques, *loop tiling* and *loop unrolling*, in an irregular kernel code example (spMxV). We have selected these techniques because we have found that the powerful loop tiling method slightly improve the performance of the object code, while, a much simpler technique, loop unrolling, is able to significantly reduce the execution time of the code.

To simplify our analysis, from now on we will focus our attention on the sparse computation spMxV and compressed data storages. In this paper, we have considered the **CRS** (*Compressed Row Storage*) and **CCS** (*Compressed Column Storage*) formats, which do not restrict our range of application nor store any unnecessary element. Basically, CRS (CCS) permits to represent the sparse matrix using three arrays (DA, CO and RO). For CRS, the first array stores the non-zero values of the matrix as they are traversed in a row-wise fashion, the second array retains the column index of each non-zero element in A, and the latter array marks the beginning of the data for each matrix row. For CCS the elements are stored as traversed by columns.

2.1 Sparse Tiling

Loop tiling is a well known loop transformation that can be used automatically by the compiler to create block algorithms and to exploit locality. It alters the way in which individual iterations are executed so that iterations from outer loops are carried out before completing all the inner loop iterations. The use of tiling with sparse matrices is not as easy. Compressed representations make difficult both the selection of the tile size and the code transformation to divide the iteration space.

When using the CRS format, the problem is that while the column coordinate of a non-null $DA(j)$ is stored in the array entry $CO(j)$, the row number is not stored in $RO(j)$. As the RO array stores a list of indices pointing to some compressed array cells, the block tiling will require the modification of this array to visit the entries by blocks. This transformation can be done during the matrix reading. Fig. 1.a sketches the tiled code for the spMxV kernel. The benefit in performance on the spMxV kernel is expected to be not very high. While arrays Y, DA, RO and CO are traversed linearly, array X is accessed more randomly. Thus, tiling can only improve locality in that array X, but with the side effect of reducing locality access to array Y.

```
rr1 = 1
 DO j = 1, n/B
      DO i = 1, n
          rr2 = NRO(i+1+n*(j-1))
          rr3 = rr2-rr1
           DO rr4 = 0,rr3-1
                Y(i)+= NDA(rr1+rr4)*X(NCO(rr1+rr4))
           ENDDO
          rr1 = rr2
      ENDDO
 ENDDO
```

(a)

```
rr1 = 1
 DO i = 1, n
      rr2 = RO(i+1)
      rr3 = rr2-rr1
      rr4 = 0
       DO WHILE (rr4 .LT. rr3)
           Y(i) = Y(i)+DA(rr1+rr4)*X(CO(rr1+rr4))
           rr4 = rr4+1
            DO WHILE (rr4 .LT. rr3-5)
                 Y(i) += DA(rr1+rr4)*X(CO(rr1+rr4))
                 Y(i) += DA(rr1+rr4+1)*X(CO(rr1+rr4+1))
                 Y(i) += DA(rr1+rr4+2)*X(CO(rr1+rr4+2))
                 Y(i) += DA(rr1+rr4+3)*X(CO(rr1+rr4+3))
                 Y(i) += DA(rr1+rr4+4)*X(CO(rr1+rr4+4))
                 rr4 = rr4+5
            ENDDO
       ENDDO
      rr1 = rr2
 ENDDO
```

(b)

Fig. 1. (a) SpMxV after the sparse tiling using the modified CRS representation; (b) SpMxV after the sparse loop unrolling of size 5 ($\delta = 5$)

2.2 Sparse Unrolling

This technique cannot be directly applied in sparse compressed representations, because the number of non-null entries per dimension is not known at compile-time. In the spMxV code the inner loop uses the index array *RO* to traverse the non-nulls of a matrix row. As the content of this array is unknown during the compilation, the optimal unrolling step should be selected depending on matrix features, as matrix homogeneity. As this kind of information is not known by the compiler, we can do it manually, as no data dependence relation is violated by the transformation in any case.

Fig. 1.b shows the spMxV kernel code with the inner loop unrolled by an example factor of $\delta = 5$. Two **DO WHILE** appear instead of the inner loop j. The inner while loop iterates a block of 5 consecutive sentences of the original j loop, while the outer while loop is used to execute the residual number of sentences. An important parameter of this transformation is the selection of the unrolling factor δ. Its value depends on sparse matrix properties, as its size, the sparsity pattern and the amount of non-null entries by row. Other facts are related to properties of the machine processor, as the amount of internal CPU registers or its ILP (Instruction Level Parallelism) capacity.

3 Experimental Results and Conclusions

In this section we present some experimental results for the codes presented before, conducted on different hardware platforms and using different compilers. We only discuss here results for the unrolling transformation, because sparse loop tiling for the spMxV kernel is predicted to have a small effect on its performance.

We have evaluated the unrolled spMxV code on three platforms: a Digital AlphaServer 4100 with a 400 MHz Alpha 21164 processor; a SGI Origin2000 with a 195 MHz MIPS R10000 processor; and a Cray T3E with a 450 MHz Alpha 21164 processor. For the purposes of an experimental validation, we run

Matrix			Alpha 21164			R10000			Cray T3E		
Name	Size	Density	δ	spMxV	**Improv.**	δ	spMxV	**Improv.**	δ	spMxV	**Improv.**
Psmigr3.rua	3140x3140	5.51%	14	22.80	**35%**	15	91.64	**22%**	6	18.82	**62.7%**
Fidapm37.rua	9152x9152	0.91%	14	32.51	**32%**	14	116.01	**45%**	6	19.70	**73.7%**
Beaflw.rra	497x507	21.2%	11	1.66	**99%**	12	13.48	**34%**	6	1.27	**79.4%**
Af23560.rua	17281x17281	0.18%	9	20.52	**23%**	3	363.1	**11%**	6	13.03	**34.7%**
S3dkq4m2.dat	90449x90449	0.03%	12	114.9	**30%**	3	1371.2	**8%**	4	54.61	**29.7%**

Fig. 2. Improvement of the spMxV kernel on the 3 platforms with different sparse matrices and different δ values (time in milliseconds)

the sparse Conjugate Gradient (CG) algorithm, the oldest, best known, and most effective of the non-stationary iterative methods for the solution of symmetric positive definite systems. Since the features of the input matrix become paramount in the algorithm behavior, we have selected a set of very different matrices from the Harwell-Boeing Collection (HB).

Fig. 2 shows the execution times for the spMxV kernel code and the improvement of the sparse unrolling, for different HB sparse matrices and using different compilers. The unrolling factor δ in table corresponds to the value with the best performance improvement. This value has a large variation range because it depends strongly on input sparse data (sparsity pattern).

Our main conclusions from this work are that tiling requires a high cost preprocessing stage to modify the storage format, an extra memory cost (NRO vector is bigger than RO), and the locality exploited in the source vector X is missed in destination (vector Y). We can not exploit temporal locality because the sparse matrix and the dense vector are linearly stored, and then does not exist any data reusage in cache line. Another conclusion is that unrolling reduces the loop overhead and obtain in most real situations a significant improvement in performance. The selection of the unrolling factor (δ) depends on the matrix sparsity pattern and size and the processor internal characteristics.

References

1. S. Carr, K.S. Mckinley and C. Tseng, *Compiler Optimizations for Improving Data Locality*, 6th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1994.
2. M. Kandemir and J. Ramanujam, *Data Relaton Vectors: A New Abstraction for Data Optimizations*, IEEE Transactions on Computers, Vol. 50, No. 8, August 2001.
3. M. O'Boyle and P. Knijnenburg, *Integrating Loop and Data Transformations for Global Optimizations*, IEEE International Conference on Parallel Architectures and Compilation Techniques, Paris, France, October 1998.
4. G. Rivera and C-W. Tseng, *Data Transformations for Eliminating Conflict Misses*, ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998.
5. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison–Wesley Pub., Redwood City, CA, 1996.